MAINFRAME COBOL

Interview Preparation & Concept Study Guide

Covering Program Structure, Data Definition, I/O, Conditional & Control Logic,
File Handling, Memory Optimization, Sub-Programs, String Handling and Arrays/Tables

Prepared for: 4+ Year Mainframe Developer & Production Support Engineer Roles

Table of Contents

Introduction: The Mainframe & COBOL Ecosystem

Definition

A mainframe is a large-scale enterprise computing platform used by banks, insurers, airlines, and retailers to store and process very high volumes of transactional data reliably, securely, and around the clock. COBOL (Common Business Oriented Language) is the primary programming language used on mainframes to implement business logic.

Concept Explanation

Mainframe processing is broadly divided into two categories:

- Batch Processing – large volumes of data are processed together, without user interaction, usually scheduled and executed through JCL (Job Control Language).
- Online Processing – data is processed in real time in response to a user action, typically through a screen/transaction managed by CICS.

A typical mainframe application stack is built from four core technologies, each responsible for a distinct concern:

- COBOL – implements the business logic (calculations, validations, decision rules).
- JCL – controls execution of batch programs (which program to run, which files/datasets to use, in what sequence).
- DB2 – relational database used to permanently store and query structured business data.
- CICS – transaction processing monitor that executes COBOL programs online, in response to screen/terminal requests.

Batch Application Stack

```
[ 1 ]  COBOL Program – contains the business logic
          |
          ▼
[ 2 ]  JCL Job – submits and executes the COBOL load module
          |
          ▼
[ 3 ]  DB2 / Flat File – stores the input and output data
```

Online Application Stack

```
[ 1 ]  User enters data on a CICS screen (BMS map)
          |
          ▼
[ 2 ]  CICS transaction invokes the COBOL program
          |
          ▼
[ 3 ]  COBOL program applies business logic and reads/writes DB2
```

```
        |
        ▼
[ 4 ]  CICS sends the response back to the screen
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between batch processing and online processing on a mainframe?
2. What are the four core technologies typically found in a mainframe application, and what role does each play?
3. Why do banks and insurance companies still rely on mainframes and COBOL for critical processing?
4. What is the difference between a COBOL program compiled for batch use versus one compiled for CICS use?
5. How does a batch COBOL program get executed end-to-end, from submission to completion?
6. What is your typical day-to-day role as a mainframe developer or production support engineer?

Topic 1: COBOL Program Structure

Division, Section & Paragraph

Definition

COBOL programs are organized in a strict, top-down hierarchical structure: Division → Section → Paragraph → Statement. Every COBOL source program is built from up to four Divisions, coded in a fixed sequence, each serving a distinct purpose.

Concept Explanation

The four Divisions of a COBOL program are:

1. IDENTIFICATION DIVISION – Provides in-program documentation. Only the PROGRAM-ID paragraph is mandatory; AUTHOR, DATE-WRITTEN, DATE-COMPILED, INSTALLATION and SECURITY are optional documentation paragraphs. Most production shops require at minimum PROGRAM-ID, AUTHOR and DATE-WRITTEN.
2. ENVIRONMENT DIVISION – Defines the external resources (files) the program uses through the INPUT-OUTPUT SECTION and FILE-CONTROL paragraph. This Division is optional, and is required only when the program directly reads/writes a physical file with OPEN/READ/WRITE (it is not required for CICS programs even when they use files, and not required for pure DB2-only programs).
3. DATA DIVISION – Defines every variable used by the program, organized into the FILE SECTION (record layouts for files defined in the Environment Division), WORKING-STORAGE SECTION (program variables and counters that retain their value across the run), and LINKAGE SECTION (variables received from a calling program or from a JCL PARM). This Division is optional if the program uses no variables at all, which in practice never happens.
4. PROCEDURE DIVISION – Contains the actual program logic, written as developer-defined Sections and Paragraphs made up of executable statements. This is where 80% of development and debugging time is spent. COBOL itself does not provide any predefined paragraph names here; naming and structuring this Division is entirely up to the developer.

Coding Line Layout (COBOL Reference/Coding Sheet Format)

Traditional fixed-format COBOL source uses column positions 1 through 80 on every line:

- Columns 1–6: Sequence number area (historically used for card sequencing; usually left blank or auto-generated today).
- Column 7: Indicator area – an asterisk (*) here marks the entire line as a comment.
- Columns 8–11: Area A – Division headers, Section headers, Paragraph names, and 01-level (and 77-level) data items must begin here.
- Columns 12–72: Area B – all executable statements and subordinate data item definitions are coded here.
- Columns 73–80: Identification/change-tag area, often used in projects to tag a change ticket number.

Example

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CBLPGM01.
       AUTHOR.      A KUMAR.
       DATE-WRITTEN. 01-JUL-2026.

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.

       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       LINKAGE SECTION.

       PROCEDURE DIVISION.
       000-MAIN-PARA.
           DISPLAY 'HELLO ALL'.
           STOP RUN.
```

Flowchart: How a COBOL Program Is Structured

```
[ 1 ]  IDENTIFICATION DIVISION – identifies the program (PROGRAM-ID mandatory)
         |
         ▼
[ 2 ]  ENVIRONMENT DIVISION – declares files used (optional, only if files are
used)
         |
         ▼
[ 3 ]  DATA DIVISION – declares all variables and record layouts (optional)
         |
         ▼
[ 4 ]  PROCEDURE DIVISION – contains the executable business logic (mandatory)
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What are the four Divisions in a COBOL program, and what is the purpose of each?
2. Which Division and which paragraph within it is mandatory in every COBOL program?
3. When is the ENVIRONMENT DIVISION required, and when can it be skipped?
4. What is the difference between the FILE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION?
5. Explain the significance of Area A and Area B in traditional fixed-format COBOL coding.
6. How are Sections and Paragraphs different from Divisions, and who defines them in the

PROCEDURE DIVISION?

7. What happens if PROGRAM-ID is missing from a COBOL program?

Topic 2: Variable Declarations

Level Numbers, PIC Clause, Data Types & Size

Definition

Variable declaration in COBOL is done in the DATA DIVISION using level numbers to define hierarchy, a data-name to identify the variable, and a PICTURE (PIC) clause to define its data type and size.

Concept Explanation

Level Numbers

Level numbers establish the hierarchy of data items:

- 01 – Record-level; the highest level used for a standalone or top-level group item. Must begin in Area A.
- 02–49 – Subordinate elementary or group items, used to build a hierarchical record structure (e.g., a group item at level 03 with elementary items at level 05 beneath it).
- 66 – RENAMES clause, used to regroup existing elementary items under a new name without allocating new memory.
- 77 – Independent, standalone elementary item that cannot be subdivided and does not belong to any group.
- 88 – Condition-name (level), used to associate a readable name with one or more specific values of its parent field; it does not occupy any memory of its own.

PIC Clause and Data Types

The PIC clause tells the compiler what type of data a variable can hold and how many bytes to allocate. COBOL supports three basic data categories:

5. Numeric (9) – digits 0–9 only, used for values involved in arithmetic. Example: PIC 9(03) stores a 3-digit unsigned number (000–999). Add an S for a signed number, e.g., PIC S9(03). Add V for an assumed (implied) decimal point, e.g., PIC S9(03)V9(02) stores a 5-digit signed number with 2 decimal places, occupying 5 bytes in DISPLAY format (the V itself takes no storage).
6. Alphabetic (A) – letters A–Z and spaces only. Example: PIC A(04). Rarely used in real projects because names, cities, and codes often contain numbers or special characters.
7. Alphanumeric (X) – any character: digits, letters, or special characters. Example: PIC X(07). This is the type most commonly used in production code, including for fields like PIN codes that are numeric-looking but never take part in arithmetic.

Choosing 9 vs X (Practical Rule)

A field should be declared numeric (9) only if it will be used in a calculation (ADD, COMPUTE, SUBTRACT, etc.). If a value merely looks numeric but is never calculated on — such as an account number, mobile number, PIN code, or claim number — it should be declared PIC X. This is because IDs that are purely numeric can overflow (e.g., a 4-digit account number reaching 9999 cannot simply be incremented), so real systems typically build such identifiers as

an alphanumeric combination of a branch/region code plus a numeric sequence, e.g., a 4-character branch code (X(04)) followed by a 6-digit sequence number (9(06)).

Group vs Elementary Items

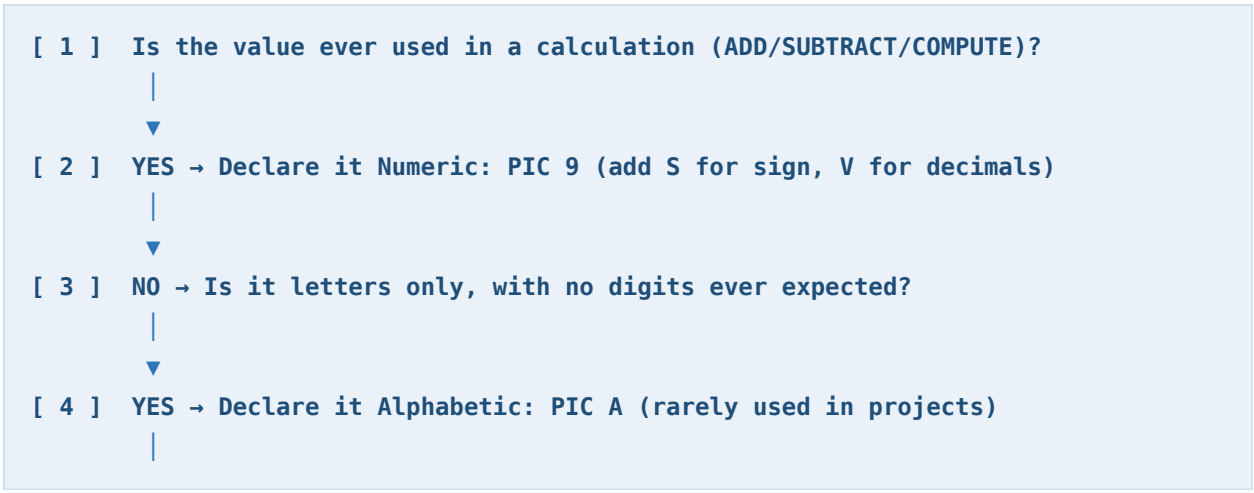
An elementary item has a PIC clause and holds actual data. A group item has no PIC clause of its own; it is simply a named collection of the elementary items beneath it, and its size and content are the sum of its subordinate items.

Example

```
       01  ACCOUNT-DETAILS.
           03  ACCOUNT-NUMBER.
               05  BRANCH-CODE      PIC X(04).
               05  ACCT-SEQ-NUM     PIC 9(06).
           03  CUST-NAME            PIC X(20).
           03  CUST-ADDRESS.
               05  CITY             PIC X(15).
               05  STATE            PIC X(15).
               05  PINCODE          PIC X(06).
           03  ACCOUNT-BALANCE      PIC S9(09)V9(02).
           03  ACCOUNT-TYPE         PIC X(01).
               88  SAVINGS-ACCT     VALUE 'S'.
               88  CURRENT-ACCT     VALUE 'C'.
           03  ACCOUNT-OPEN-DATE.
               05  YYYY             PIC 9(04).
               05  MM               PIC 9(02).
               05  DD               PIC 9(02).

       77  TOTAL-ACCOUNT-BALANCE    PIC S9(11)V9(02).
```

Here, ACCOUNT-DETAILS is a group item (its size is the sum of all fields under it). ACCOUNT-NUMBER is itself a group made up of BRANCH-CODE (alphanumeric, since branch codes like 'A001' are never calculated on) and ACCT-SEQ-NUM (numeric, since it increases by 1 for each new customer). ACCOUNT-TYPE uses an 88-level to make the code readable.

Flowchart: Deciding How to Declare a Field


[1] Is the value ever used in a calculation (ADD/SUBTRACT/COMPUTE)?
[2] YES → Declare it Numeric: PIC 9 (add S for sign, V for decimals)
[3] NO → Is it letters only, with no digits ever expected?
[4] YES → Declare it Alphabetic: PIC A (rarely used in projects)

```
        ▼
[ 5 ]  NO → Declare it Alphanumeric: PIC X (default choice for IDs/codes/text)
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between level 01, level 77, and level 88 in COBOL?
2. What does the PIC clause define, and what are the three basic COBOL data types?
3. What is the difference between PIC 9, PIC S9, and PIC 9V9? How many bytes does S9(05)V9(02) occupy in DISPLAY format?
4. Why would a developer choose PIC X over PIC 9 for a field like Account Number or PIN Code even though it 'looks' numeric?
5. What is the difference between a group item and an elementary item?
6. What is the purpose of an 88-level condition name, and does it consume any memory?
7. What is the maximum number of digits you can define in a PIC 9 clause in standard COBOL, and how would you store a larger value?

Topic 3: Input-Output & Arithmetic

VALUE Clause, ACCEPT & DISPLAY, COMPUTE

Definition

Input-output statements let a COBOL program receive data from and send data to the outside world, while arithmetic statements (ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE) perform calculations on that data. The VALUE clause initializes a variable at the point of declaration in the DATA DIVISION.

Concept Explanation

Ways Data Enters a Program

- VALUE clause — initializes a variable at compile time, within the program itself.
- ACCEPT statement — reads data supplied at run time through JCL, typically via the SYSIN DD * dataset.
- READ statement — reads data from a file (covered under File Handling).
- MOVE statement — assigns data from one variable, or a literal, to another variable within the program.
- SELECT query — retrieves data from a DB2 table (embedded SQL).
- CALL … USING — receives data passed in from a calling program.
- LINKAGE SECTION with PARM — receives a run-time parameter passed from the JCL EXEC statement.
- RECEIVE command — receives data entered on a CICS online screen.

Ways Data Leaves a Program

- DISPLAY statement — writes to the job's SYSOUT/spool (temporary, for logging/debugging, not permanent output).
- WRITE statement — writes a record to a file (covered under File Handling).
- INSERT/UPDATE — writes to a DB2 table (embedded SQL).
- SEND command — sends a response to a CICS online screen.

VALUE Clause

A newly allocated variable's memory can contain leftover/unpredictable data, so it must be initialized before use. The VALUE clause sets this starting value at compile time:

```
       01  VAR1              PIC 9(04)  VALUE ZERO.
       01  VAR2              PIC X(04)  VALUE SPACE.
       01  SWITCH-VARIABLE   PIC X(01)  VALUE 'N'.
       01  CLAIM-STATUS      PIC X(10)  VALUE 'APPROVED'.
```

ACCEPT and DISPLAY

ACCEPT reads a value supplied through the SYSIN DD * card in the job's JCL, one value per ACCEPT statement, in the sequence they are coded. DISPLAY prints a literal message and/or

the value of a variable to SYSOUT. In production code, DISPLAY is mainly used for audit trails, counts, and error/abend diagnostics — it is not used to output business data, because SYSOUT is a temporary spool location, not a permanent data store.

```
       ACCEPT WS-CUST-NAME.
       DISPLAY 'CUSTOMER NAME IS: ' WS-CUST-NAME.

//SYSIN    DD *
ANIL KUMAR
/*
```

Arithmetic — ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE

COBOL provides individual verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE) for simple single-operation arithmetic, and the COMPUTE statement for any combination of arithmetic in a single formula, following standard operator precedence.

```
       ADD  WS-NUM1 TO WS-NUM2 GIVING WS-TOTAL.

       COMPUTE WS-TOTAL-DUE =
               WS-PREV-MONTH-DUE - WS-PRV-AMT-PAID
             + WS-CURRENT-AMT-DUE
             + (WS-PREV-MONTH-DUE - WS-PRV-AMT-PAID) * 0.18
           ON SIZE ERROR
               DISPLAY 'ERROR: SIZE OVERFLOW IN TOTAL-DUE CALC'
       END-COMPUTE.
```

The ON SIZE ERROR clause is important in production programs: if a receiving field is too small for the calculated result (for example, TOTAL PIC 9(02) cannot hold NUM1 (50) + NUM2 (50) = 100), the result would silently truncate unless ON SIZE ERROR traps the overflow. Since a plain DISPLAY of an error is easy to miss in a job with thousands of daily runs, production programs typically also set a non-zero RETURN-CODE or force an ABEND so that Production Support is automatically alerted rather than relying on someone manually reading the job log.

Flowchart: Program Data Flow

```
[ 1 ]  Get the data into the program (VALUE / ACCEPT / READ / CALL-USING)
         |
         ▼
[ 2 ]  Process the data as per business logic (MOVE / IF / COMPUTE)
         |
         ▼
[ 3 ]  Send the processed data out (DISPLAY / WRITE / INSERT / SEND)
         |
         ▼
[ 4 ]  Repeat for the next record until end-of-input
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the purpose of the VALUE clause, and why must WORKING-STORAGE variables typically be initialized?
2. How does ACCEPT receive data at run time, and which JCL DD statement supplies it?
3. Why is DISPLAY not considered a proper way to output permanent business data in a production COBOL program?
4. What is the difference between using individual arithmetic verbs (ADD/SUBTRACT) versus COMPUTE?
5. What does the ON SIZE ERROR clause do, and why is it important in a COMPUTE statement?
6. In a production support scenario, how would you detect that a calculation silently truncated instead of erroring out?
7. What is the difference between DISPLAY output, a non-zero RETURN-CODE, and an ABEND as ways of signaling a problem to production support?

Topic 4: Data Movement Verbs

Simple Move, Move with OF Clause, Group Move & MOVE CORRESPONDING

Definition

The MOVE statement copies data — a literal constant or the contents of one variable — into one or more receiving variables. It is one of the three verbs (MOVE, IF, PERFORM) that make up the majority of statements in any real COBOL program.

Concept Explanation

Justification Rules

- Numeric data is right-justified when moved: if the receiving field is larger, the value is zero-padded on the left; if smaller, high-order (leftmost) digits are truncated.
- Alphanumeric data is left-justified when moved: if the receiving field is larger, it is space-padded on the right; if smaller, low-order (rightmost) characters are truncated.

MOVE does not raise any warning or error on truncation the way COMPUTE/ADD can with ON SIZE ERROR — the only protection is defining the receiving field with an adequate size in the first place.

Types of MOVE

8. Simple MOVE — moves a single literal or variable to one or more uniquely-named receiving fields. Example: MOVE 10 TO WS-A, WS-B, WS-C. A statement like MOVE WS-A TO WS-A is invalid because the source and target must be distinct.
9. Group MOVE — moves an entire group item to another group item in a single statement, provided the two groups are 'identical': same number of subordinate fields, in the same sequence, with the same data types/sizes. This performs a byte-for-byte copy without regard to the individual field boundaries.
10. MOVE CORRESPONDING (MOVE CORR) — moves data between two group items field-by-field, matching subordinate fields by name rather than position, so it still works correctly even when the fields are declared in a different sequence in the source and target groups. In most production shops MOVE CORR (and plain Group MOVE) are avoided in favor of explicit, field-by-field MOVE statements, because they make the code harder to trace during maintenance and support.
11. MOVE … OF … — used to move one specific elementary field that is qualified by its parent group name, when the same field name is reused across multiple group items (e.g., CLAIM-NUMBER exists in both CLAIM-DETAILS-IN and CLAIM-DETAILS-OUT). The OF clause disambiguates which occurrence is meant.

Example

```
       01  CLAIM-DETAILS-IN.
           03  CLAIM-NUMBER      PIC X(08).
           03  CLAIM-AMOUNT      PIC 9(07)V99.

       01  CLAIM-DETAILS-OUT.
```

```
          03  CLAIM-NUMBER      PIC X(08).
          03  CLAIM-AMOUNT      PIC 9(07)V99.

       01  DATE-YMD.                    01  DATE-DMY.
           03  YYYY  PIC 9(04).             03  DD    PIC 9(02).
           03  MM    PIC 9(02).             03  MM    PIC 9(02).
           03  DD    PIC 9(02).             03  YYYY  PIC 9(04).

      * -- Simple move (unique names) -----------------
           MOVE ZERO  TO WS-TOTAL-COUNT.
           MOVE SPACE TO WS-ERROR-MSG.

      * -- Move OF clause (same field name reused) -----
           MOVE CLAIM-NUMBER OF CLAIM-DETAILS-IN
             TO CLAIM-NUMBER OF CLAIM-DETAILS-OUT.

      * -- Group move (identical layouts) ---------------
           MOVE CLAIM-DETAILS-IN TO CLAIM-DETAILS-OUT.

      * -- Move CORRESPONDING (fields in different order)
           MOVE CORRESPONDING DATE-YMD TO DATE-DMY.
```

Flowchart: Choosing the Right MOVE

```
[ 1 ]  Are you moving one field with a unique, unqualified name?
          |
          ▼
[ 2 ]  YES → Use a Simple MOVE
          |
          ▼
[ 3 ]  NO → Does the same field name exist in more than one group?
          |
          ▼
[ 4 ]  YES → Use MOVE fieldname OF groupname
          |
          ▼
[ 5 ]  NO → Are the two groups identical in layout and sequence?
          |
          ▼
[ 6 ]  YES → Use a Group MOVE   |   NO → Use MOVE CORRESPONDING
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between how MOVE handles numeric versus alphanumeric data (justification and padding)?

2. What happens to a MOVE statement when the receiving field is too small for the source data?
3. What conditions must be true for a Group MOVE to work correctly between two group items?
4. How does MOVE CORRESPONDING differ from a plain Group MOVE?
5. Why do many production shops discourage the use of MOVE CORRESPONDING or Group MOVE?
6. When would you need to use the OF clause with a MOVE statement, and why?
7. Does COBOL raise a runtime error on data truncation during a MOVE the way it can for COMPUTE? How would you prevent truncation?

Topic 5: Conditional Statements

IF & EVALUATE — CONTINUE & NEXT SENTENCE

Definition

Conditional statements let a program execute or bypass sections of logic depending on whether a condition evaluates to TRUE or FALSE. COBOL provides the IF statement for general-purpose branching and the EVALUATE statement for cleanly handling multiple, mutually exclusive conditions.

Concept Explanation

IF Statement Variants

12. Simple IF — executes a block only when the condition is TRUE; if FALSE, control simply skips to the statement after END-IF.
13. IF-ELSE — executes one block when TRUE and a different block when FALSE.
14. IF-ELSE Ladder (nested at the same logical level) — used to test several mutually exclusive conditions in sequence; as soon as one condition is found TRUE, the remaining conditions in the ladder are not evaluated, which is efficient. Each IF-ELSE in the ladder needs its own END-IF.
15. Nested IF — an IF statement coded entirely inside the TRUE or FALSE branch of another IF, used to test a sub-condition only when an outer condition is met.

EVALUATE Statement

EVALUATE is COBOL's structured, multi-branch conditional, comparable to a SWITCH/CASE statement in other languages. It is preferred over a long IF-ELSE ladder when there are several distinct conditions to check, because it is easier to read and maintain. It has two common forms: EVALUATE <identifier/expression> WHEN <value> (compares one item to several possible values), and EVALUATE TRUE WHEN <condition> (evaluates a series of independent Boolean conditions, functionally similar to an IF-ELSE ladder but more structured). WHEN OTHER handles any value not explicitly matched.

CONTINUE vs NEXT SENTENCE

Both are 'do nothing' statements used to explicitly exit an IF block without performing any action, but they behave differently:

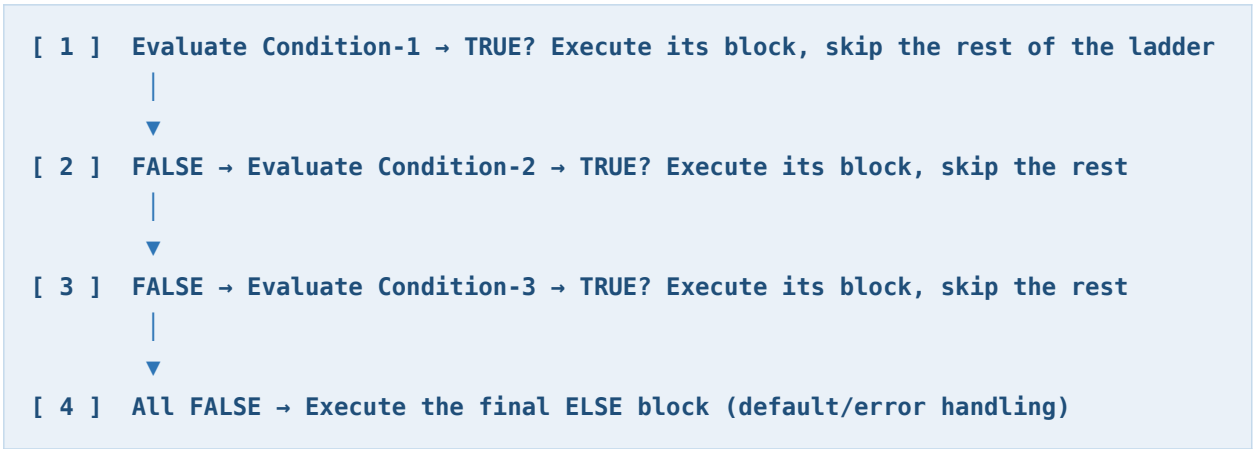
- CONTINUE — passes control to the very next statement after the nearest END-IF (i.e., it respects the scope terminator).
- NEXT SENTENCE — passes control to the next statement after the next period ('.') in the program, which may skip over several nested END-IFs if the sentence spans multiple nested conditions. Because of this behavior difference, NEXT SENTENCE is considered risky in deeply nested logic and CONTINUE is generally the safer, more predictable choice in modern COBOL.

Example

```
      * Populate a customer title based on gender, with error
      * handling for unexpected input (production-style logic).

           IF GENDER OF CUST-DETAILS-IN = 'MALE'
               MOVE 'MR'  TO CUST-TITLE OF CUST-DETAILS-OUT
           ELSE
           IF GENDER OF CUST-DETAILS-IN = 'FEMALE'
               MOVE 'MRS' TO CUST-TITLE OF CUST-DETAILS-OUT
           ELSE
               DISPLAY 'ERROR IN GENDER INPUT: ' CUST-DETAILS-IN
               PERFORM 900-ABEND-PARA
           END-IF
           END-IF.

      * The same logic written with EVALUATE:

           EVALUATE GENDER OF CUST-DETAILS-IN
               WHEN 'MALE'
                   MOVE 'MR'  TO CUST-TITLE OF CUST-DETAILS-OUT
               WHEN 'FEMALE'
                   MOVE 'MRS' TO CUST-TITLE OF CUST-DETAILS-OUT
               WHEN OTHER
                   DISPLAY 'ERROR IN GENDER INPUT: ' CUST-DETAILS-IN
                   PERFORM 900-ABEND-PARA
           END-EVALUATE.
```

Note the design choice here: an earlier, simpler version of this logic used two independent IF statements (IF MALE ... , IF FEMALE ...) without an ELSE. That version always evaluates both conditions for every record (a minor performance cost) but, more importantly, silently produces no title at all for bad input data (e.g., 'MAIL' instead of 'MALE'). The IF-ELSE-ladder / EVALUATE version trades a negligible amount of performance for correctness, by explicitly trapping and reporting invalid data — in production code, correctness of logic is prioritized over micro-level performance.

Flowchart: IF-ELSE Ladder Execution


[1] Evaluate Condition-1 → TRUE? Execute its block, skip the rest of the ladder
[2] FALSE → Evaluate Condition-2 → TRUE? Execute its block, skip the rest
[3] FALSE → Evaluate Condition-3 → TRUE? Execute its block, skip the rest
[4] All FALSE → Execute the final ELSE block (default/error handling)

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between a Simple IF, IF-ELSE, IF-ELSE ladder, and Nested IF?
2. When would you prefer EVALUATE over an IF-ELSE ladder, and why?
3. What is the difference between EVALUATE <identifier> WHEN <value> and EVALUATE TRUE WHEN <condition>?
4. Explain the difference between CONTINUE and NEXT SENTENCE with an example of nested IF statements.
5. Why can NEXT SENTENCE be risky to use in deeply nested conditional logic?
6. In a real program, why might a developer intentionally add an ELSE / WHEN OTHER clause even if it seems unnecessary for valid data?
7. How would you handle unexpected or invalid input data in a conditional block in a production program?

Topic 6: Control Sequence Verbs

PERFORM, STOP RUN, GOBACK, EXIT, EXIT PROGRAM, GO TO

Definition

Control sequence verbs alter the default top-to-bottom, statement-by-statement execution order of a COBOL program — either to loop over the same logic repeatedly (PERFORM), to jump permanently to another paragraph (GO TO), to leave a paragraph (EXIT), or to end program execution and return control to the caller (STOP RUN, GOBACK, EXIT PROGRAM).

Concept Explanation

Ending Program Execution: STOP RUN vs GOBACK vs EXIT PROGRAM

A COBOL program can be either a Main Program (executed directly by JCL) or a Sub-Program (executed when called by another COBOL program). The exit verb used depends on this role:

- STOP RUN — terminates the program and returns control all the way back to the operating system/JCL. It should only be coded in a Main Program; using it inside a Sub-Program would end the entire job, not just return to the caller.
- EXIT PROGRAM — terminates the current program and returns control to the calling (Main) program. It should only be coded in a Sub-Program.
- GOBACK — functionally similar to STOP RUN/EXIT PROGRAM but context-sensitive: it returns to the operating system if coded in a Main Program, or to the calling program if coded in a Sub-Program. Because of this flexibility, GOBACK is commonly preferred in modern coding standards over hard-coding STOP RUN or EXIT PROGRAM.

PERFORM

PERFORM transfers control to a paragraph (or a block of inline statements) to execute it, and — unlike GO TO — control automatically returns to the statement immediately after the PERFORM once that logic completes. This makes PERFORM the basis for both structured subroutine calls within a program and for looping.

16. Inline PERFORM — the logic is coded directly between PERFORM and END-PERFORM. It cannot be reused/invoked from elsewhere in the program.
17. Out-of-line PERFORM (PERFORM paragraph-name) — the logic lives in a separate paragraph, and can be invoked (reused) from multiple places in the program.
18. PERFORM paragraph-name N TIMES — executes the paragraph a fixed number of times.
19. PERFORM paragraph-name UNTIL condition — loops, executing the paragraph repeatedly, checking the condition before each iteration (test-before-perform); the loop stops as soon as the condition becomes TRUE.
20. PERFORM para-1 THRU para-2 — executes a contiguous range of paragraphs together as a single logical unit, useful for grouping several small paragraphs under one call.
21. PERFORM VARYING — a counter-controlled loop that initializes a variable, tests a condition, and increments the variable automatically each pass (used heavily with OCCURS tables — see Topic 11).

GO TO and EXIT

- GO TO paragraph-name — unconditionally transfers control to another paragraph, but unlike PERFORM, control does NOT return automatically. Because this can make program flow hard to follow ('spaghetti code'), GO TO is generally avoided in modern structured COBOL in favor of PERFORM.
- EXIT — a no-operation statement used purely as a placeholder to mark the end of a paragraph, commonly used as the target/end-point paragraph in a PERFORM … THRU …-EXIT construct (e.g., 300-PROCESS-PARA THRU 300-EXIT).

Example — Standard Top-Down Program Skeleton

```
       PROCEDURE DIVISION.
       000-MAIN-PARA.
           DISPLAY 'PGMNAME STARTED'.
           PERFORM 100-INITIAL-PARA    THRU 100-EXIT.
           PERFORM 200-GET-INPUT-PARA  THRU 200-EXIT.
           PERFORM 300-PROCESS-PARA    THRU 300-EXIT
                   UNTIL WS-END-OF-INPUT = 'Y'.
           DISPLAY 'TOTAL RECORDS READ: ' WS-INPUT-COUNT.
           STOP RUN.

       100-INITIAL-PARA.
           MOVE ZERO  TO WS-INPUT-COUNT WS-OUTPUT-COUNT.
           MOVE 'N'   TO WS-END-OF-INPUT.
           OPEN INPUT  CLAIM-FILE.
           OPEN OUTPUT REPORT-FILE.
       100-EXIT.
           EXIT.

       200-GET-INPUT-PARA.
           READ CLAIM-FILE
               AT END MOVE 'Y' TO WS-END-OF-INPUT
           END-READ.
       200-EXIT.
           EXIT.

       300-PROCESS-PARA.
           ADD 1 TO WS-INPUT-COUNT.
           PERFORM 320-CALCULATE-PARA   THRU 320-EXIT.
           PERFORM 340-WRITE-OUTPUT-PARA THRU 340-EXIT.
           PERFORM 200-GET-INPUT-PARA    THRU 200-EXIT.
       300-EXIT.
           EXIT.
```

Flowchart: PERFORM UNTIL Loop

```
[ 1 ]  Initialize counter/switch (e.g., WS-END-OF-INPUT = 'N')
         |
         ▼
```

```
[ 2 ]  PERFORM ... UNTIL condition → Test condition FIRST
          |
          ▼
[ 3 ]  Condition FALSE → Execute paragraph body, then re-test condition
          |
          ▼
[ 4 ]  Condition TRUE  → Exit the loop, continue with the next statement
```

Flowchart: JCL-to-Program Execution & Exit Path

```
[ 1 ]  JCL is submitted → JES (Job Entry Subsystem) checks syntax
          |
          ▼
[ 2 ]  No JCL error → JES loads the compiled Load Module and passes control to the
OS
          |
          ▼
[ 3 ]  OS executes the Main Program (PGMA)
          |
          ▼
[ 4 ]  PGMA issues a CALL to Sub-Program (PGMB) → PGMB runs and hits GOBACK/EXIT
PROGRAM
          |
          ▼
[ 5 ]  Control returns to PGMA → PGMA completes and hits STOP RUN/GOBACK → control
returns to the OS
          |
          ▼
[ 6 ]  OS notifies JES of completion → Condition Code (e.g., MAXCC=0000) is
reported
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between STOP RUN, GOBACK, and EXIT PROGRAM, and where should each be used?
2. Why is GOBACK often preferred over hard-coding STOP RUN or EXIT PROGRAM in reusable modules?
3. What is the fundamental difference between PERFORM and GO TO in terms of control flow?
4. What is the difference between an inline PERFORM and an out-of-line PERFORM?
5. How does PERFORM ... UNTIL evaluate its condition — before or after each iteration — and why does that matter?

6. What is the purpose of PERFORM ... THRU ...-EXIT, and why is EXIT used as the terminating paragraph?
7. Why is GO TO discouraged in modern structured COBOL programming?
8. Describe, end-to-end, what happens from JCL submission to a batch job completing successfully.

Topic 7: File Handling

OPEN, READ, WRITE, REWRITE, DELETE & CLOSE

Definition

File handling verbs allow a COBOL program to read data from, and write data to, physical datasets (files) defined in the JCL. Every file used by a batch COBOL program must first be declared in the ENVIRONMENT DIVISION (FILE-CONTROL, with a SELECT/ASSIGN clause) and described in the DATA DIVISION FILE SECTION (FD entry with its record layout), before it can be opened and processed in the PROCEDURE DIVISION.

Concept Explanation

The File Handling Verbs

22. OPEN — makes a file available for processing. A file can be opened as INPUT (read-only), OUTPUT (write-only, creates/overwrites), I-O (both read and write, needed for REWRITE/DELETE), or EXTEND (append to an existing file). A record area cannot be referenced before its file is opened — attempting to do so is a common cause of an S0C4 abend.
23. READ — retrieves the next sequential (or, for indexed/relative files, a specific keyed) record from an input file into its record area. The AT END clause specifies what to do when there are no more records — this is how end-of-file is detected in COBOL (there is no separate 'is file empty' check).
24. WRITE — adds a new record to an output file from its record area.
25. REWRITE — updates an existing record in place on a file opened as I-O; the record must already exist (it replaces a record with the same key/position), unlike WRITE which adds a new one.
26. DELETE — removes an existing record from a file opened as I-O (applicable to indexed/relative files, not sequential ones, since sequential files cannot have records removed from the middle).
27. CLOSE — releases the file when processing is finished. Every file that is OPENed must be CLOSEd; a batch job that fails to close a file may lose buffered output or leave the dataset in an inconsistent state.

How a File Is Defined End-to-End

A file must be tied together across three layers: the JCL DD statement (physical dataset name, DCB attributes such as LRECL/RECFM/BLKSIZE), the COBOL SELECT clause (ASSIGN TO the DD name), and the COBOL FD/01-level record layout (which must match the file's physical record length).

Example — JCL and COBOL Working Together

```
//STEP1    EXEC PGM=FILEPGM1
//STEPLIB  DD DSN=IBMUSER.T8.LOAD,DISP=SHR
//ACCTDD01 DD DSN=IBMUSER.T8.ACCT.FILE,DISP=(NEW,CATLG,DELETE),
```

```
//            UNIT=3390,VOL=SER=JASYS1,
//            SPACE=(TRK,(1,1),RLSE),
//            DCB=(DSORG=PS,LRECL=30,RECFM=FB,BLKSIZE=0)
```

```
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT ACCT-FILE ASSIGN TO ACCTDD01.

       DATA DIVISION.
       FILE SECTION.
       FD  ACCT-FILE
           RECORDING MODE IS F.
       01  ACCT-RECORD.
           05  ACCT-NUM       PIC X(05).
           05  ACCT-NAME      PIC X(10).
           05  ACCT-BALANCE   PIC 9(05).
           05  ACCT-TYPE      PIC X(10).

       PROCEDURE DIVISION.
       000-MAIN-PARA.
           OPEN OUTPUT ACCT-FILE.
           PERFORM 200-GET-INPUT THRU 200-EXIT.
           PERFORM 300-PROCESS THRU 300-EXIT
                   UNTIL WS-END-OF-INPUT = 'Y'.
           CLOSE ACCT-FILE.
           STOP RUN.

       200-GET-INPUT.
           ACCEPT ACCT-NUM.
           IF ACCT-NUM = SPACE
               MOVE 'Y' TO WS-END-OF-INPUT
           END-IF.
       200-EXIT. EXIT.

       300-PROCESS.
           WRITE ACCT-RECORD.
           PERFORM 200-GET-INPUT THRU 200-EXIT.
       300-EXIT. EXIT.
```

Flowchart: Standard File-Processing Pattern

```
[ 1 ]  OPEN the file(s) (INPUT / OUTPUT / I-O)
        |
        ▼
[ 2 ]  READ the first record  →  AT END, set an end-of-file switch
        |
        ▼
[ 3 ]  PERFORM ... UNTIL end-of-file switch = 'Y'
```

```
        |
        ▼
[ 4 ]       Process the current record (validate, calculate, WRITE/REWRITE output)
        |
        ▼
[ 5 ]       READ the next record  →  AT END, set the end-of-file switch
        |
        ▼
[ 6 ]  CLOSE the file(s) once the loop ends
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What are the different OPEN modes in COBOL, and when would you use each (INPUT, OUTPUT, I-O, EXTEND)?
2. How does COBOL detect end-of-file, and what clause is used with the READ statement for that purpose?
3. What is the difference between WRITE and REWRITE?
4. Under what file organizations can you use DELETE, and why can't you DELETE from a purely sequential file?
5. What could cause an S0C4 abend related to file handling, and how would you diagnose it from a dump?
6. What is the relationship between the JCL DD statement, the SELECT/ASSIGN clause, and the FD record layout?
7. What happens if a program does not CLOSE a file it opened for output?
8. How would you handle a mismatch between the LRECL defined in JCL and the 01-level record layout in COBOL?

Topic 8: Memory Saving

USAGE Clause, 88-Level, REDEFINES & RENAMES (66-Level)

Definition

COBOL provides several techniques to reduce the memory a program's variables consume and to make coded/compressed data easier to work with: the USAGE clause (choosing how numeric data is physically stored), REDEFINES (sharing the same memory between different variable definitions), RENAMES/66-level (regrouping existing fields under a new name), and 88-level condition names (making coded values readable).

Concept Explanation

USAGE Clause — DISPLAY, COMP, COMP-3

Numeric data can be stored in three principal formats:

28. USAGE DISPLAY (character format, the default) — each digit is stored as one full byte (its EBCDIC character representation). Simple but the least memory-efficient.
29. USAGE COMP (binary format) — the number is stored in true binary, in bits, giving fixed byte sizes: S9(01) to S9(04) → 2 bytes; S9(05) to S9(09) → 4 bytes; S9(10) to S9(18) → 8 bytes. COMP is generally used for integers, such as index/subscript or counter fields.
30. USAGE COMP-3 (packed decimal / 'packed') — the number is stored as packed hexadecimal digits, two digits per byte (a nibble each), plus one extra nibble for the sign. The formula for the number of bytes needed for S9(n) or S9(p)V9(q) where n = p+q digits is: if n is even, bytes = (n/2) + 1; if n is odd, bytes = (n+1)/2. COMP-3 is the format most commonly used for monetary and decimal fields in production file layouts, because it saves space while still supporting decimals, and it is the format most frequently referenced in interviews.

Comparative Example (S9(10))

- DISPLAY: 10 bytes
- COMP: 8 bytes
- COMP-3: 6 bytes

Beyond saving disk/memory space and improving I/O performance, COMP and COMP-3 fields also have the side benefit of being non-human-readable in a raw file browse (they appear as unreadable hex bytes rather than plain digits), which naturally protects sensitive numeric data such as balances from being casually read without the copybook layout — though this is a byproduct, not a designed security control.

REDEFINES

REDEFINES lets one or more alternate data descriptions share the exact same memory location as a previously defined item, starting at the same address and occupying the same (or less) space. It has two main uses:

- Reinterpreting the same bytes as a different breakdown — e.g., splitting a 10-byte date string 'YYYY-MM-DD' into YYYY, a filler for the dash, MM, a filler, and DD, without allocating any extra memory.
- Unpacking a table of fixed values — e.g., a block of FILLER literals initialized with VALUE clauses can be REDEFINED as an OCCURS table of error codes/messages, letting the program access the same hard-coded data through subscripts/SEARCH instead of by position.

A REDEFINES item does not itself consume additional storage; it is a second lens on the same bytes.

RENAMES (66-Level)

The 66-level RENAMES clause lets a developer group an existing set of elementary items (that may cross group boundaries in the original 01-level definition) under one new name, again without allocating any new memory. It is used far less often than REDEFINES in day-to-day project work but can appear in legacy copybooks.

88-Level Condition Names

An 88-level item associates one or more literal values with a readable condition name for its parent field. It has no PIC clause and consumes no memory of its own — its only purpose is readability and maintainability. It is not a memory-saving technique (that is a common misconception); the actual memory saving in a coded-status scenario comes from storing a single-character code (PIC X(01)) instead of a full descriptive string (e.g., PIC X(15) for 'OUT FOR DELIVERY'), and the 88-level is what makes testing that code readable in the PROCEDURE DIVISION.

Example

```
      * ---------- REDEFINES: reinterpret a date string -------
       01  DATE-YMD             PIC X(10) VALUE '2026-07-01'.
       01  DATE-YMD-RED REDEFINES DATE-YMD.
           03  YYYY             PIC 9(04).
           03  FILLER           PIC X(01).
           03  MM               PIC 9(02).
           03  FILLER           PIC X(01).
           03  DD               PIC 9(02).

      * ---------- USAGE clause & 88-level ---------------------
       01  ORDER-STATUS-CODE    PIC X(01).
           88  ORDER-PACKED       VALUE 'P'.
           88  ORDER-SHIPPED      VALUE 'S'.
           88  ORDER-OUT-FOR-DELV VALUE 'O'.
           88  ORDER-DELIVERED    VALUE 'D'.

       01  ACCT-BALANCE-DISPLAY  PIC S9(09)V9(02).
       01  ACCT-BALANCE-COMP     PIC S9(09)V9(02) USAGE COMP.
       01  ACCT-BALANCE-COMP3    PIC S9(09)V9(02) USAGE COMP-3.

      * Usage in PROCEDURE DIVISION:
```

```
           IF ORDER-OUT-FOR-DELV
               DISPLAY 'ORDER IS ON THE WAY'
           END-IF.
```

Flowchart: Choosing a USAGE Clause

```
[ 1 ]  Is the field a monetary/decimal amount stored in a file/table? → Use COMP-3
(most common project choice)
        |
        ▼
[ 2 ]  Is the field a simple integer counter/index/subscript? → Use COMP
        |
        ▼
[ 3 ]  Does the field need to be humanly readable in a raw browse/debug listing? →
Use DISPLAY (default)
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between DISPLAY, COMP, and COMP-3 usage, both in storage mechanism and byte size?
2. How do you calculate the number of bytes required to store a COMP-3 field, e.g., S9(07)V9(02)?
3. What is REDEFINES used for, and does it allocate any additional memory?
4. Give a real scenario where you would use REDEFINES on a file layout.
5. What is the difference between REDEFINES and RENAMES (66-level)?
6. Does an 88-level item save memory? If not, what is its actual purpose?
7. Why is COMP-3 the most commonly used format for numeric fields in production file layouts?
8. How would you view/browse a file that contains COMP-3 data using a mainframe file utility such as File-AID?

Topic 9: Sub-Program

CALL Statement — Static & Dynamic Call

Definition

A Sub-Program is a separately compiled COBOL program invoked from another COBOL program (the Main Program, or another Sub-Program) using the CALL statement. This enables code reuse and reduces the complexity of any single program by splitting logic into smaller, focused modules.

Concept Explanation

Coding and using a sub-program involves four concerns: (1) how the calling program passes control to the sub-program, (2) how it passes data to the sub-program, (3) how the sub-program receives that data, and (4) how the sub-program passes control and results back.

1) Passing Control — Static vs Dynamic CALL

There are two ways to CALL a sub-program:

31. Static Call — CALL 'SUBPGM-NAME' (a literal, constant program name). At compile/link time, the sub-program's compiled object is combined with the main program's object into a single load module. If the sub-program's logic changes, both the sub-program AND the calling program must be re-compiled/re-linked, because they are bound together into one load module.
32. Dynamic Call — CALL WS-VARIABLE-NAME (the program name is held in a variable, resolved at run time). The main program and sub-program remain separate, independent load modules. If the sub-program's logic changes, only the sub-program itself needs to be re-compiled — the calling (main) program does not need to be touched.

Impact Analysis for a Chain of Calls (PGMA → PGMB → PGMC)

- If both links are Static: changing PGMC requires re-compiling PGMC, PGMB, and PGMA.
- If both links are Dynamic: changing PGMC requires re-compiling only PGMC.
- If PGMA→PGMB is Static and PGMB→PGMC is Dynamic: changing PGMC requires re-compiling only PGMC.
- If PGMA→PGMB is Dynamic and PGMB→PGMC is Static: changing PGMC requires re-compiling PGMC and PGMB.

General guidance: Static calls are preferred within the same application/team because they perform slightly better (the sub-program is already resolved at link time, with no run-time search). Dynamic calls are typically reserved for calling a sub-program owned by a different application/team — for example, an Insurance Claims application dynamically calling a Policy application's update sub-program lets the Policy team update their own sub-program's logic independently, without forcing the Claims team to re-compile their main program.

2) Passing Data — CALL … USING and the Reference/Content Distinction

Data is passed to a sub-program with the USING clause: CALL 'SUBPGM' USING VAR1, VAR2. There are two passing mechanisms:

- BY REFERENCE (the default) — the calling and called program's variables share the same memory address; any change the sub-program makes to the variable is visible back in the calling program once control returns. This is, in practice, the only way COBOL sends a result back to the caller, since COBOL has no separate 'return value' statement.
- BY CONTENT — a copy of the value is passed; the calling and called program's variables occupy different memory, so changes made inside the sub-program do NOT affect the caller's variable. This is used when a variable must be protected from being altered by the sub-program.

As a practical rule, a caller must remember to pass both its input variables and an (initialized) output variable in the USING list, since that output variable is the only channel by which a result comes back.

3) Receiving Data in the Sub-Program — LINKAGE SECTION

Every variable the sub-program expects to receive is declared in its DATA DIVISION LINKAGE SECTION (no memory is allocated here — it maps onto the caller's memory) and listed, in the exact same order as the caller's USING clause, in the sub-program's PROCEDURE DIVISION USING clause. A mismatch in the number, order, or size of parameters between the CALL...USING and the PROCEDURE DIVISION USING is a classic cause of an S0C4 (invalid memory reference) or S0C7 (invalid/bad numeric data) abend.

4) Returning Control — EXIT PROGRAM / GOBACK (never STOP RUN)

A sub-program must end with EXIT PROGRAM or GOBACK, never STOP RUN — STOP RUN would terminate the entire job rather than simply returning to the calling program (see Topic 6).

Example

```
      * -------- MAIN PROGRAM (MAINPGM) --------
       WORKING-STORAGE SECTION.
       01  WS-NUM1   PIC 9(05).
       01  WS-NUM2   PIC 9(05).
       01  WS-TOTAL  PIC 9(06).

       PROCEDURE DIVISION.
           MOVE 10   TO WS-NUM1.
           MOVE 20   TO WS-NUM2.
           MOVE ZERO TO WS-TOTAL.
           CALL 'ADDPGM01' USING WS-NUM1, WS-NUM2, WS-TOTAL.
           DISPLAY 'TOTAL IS ' WS-TOTAL.
           STOP RUN.

      * -------- SUB-PROGRAM (ADDPGM01) --------
       DATA DIVISION.
       LINKAGE SECTION.
       01  NUM1   PIC 9(05).
       01  NUM2   PIC 9(05).
       01  TOTAL  PIC 9(06).

       PROCEDURE DIVISION USING NUM1, NUM2, TOTAL.
```

```
           COMPUTE TOTAL = NUM1 + NUM2.
           GOBACK.
```

Flowchart: Control & Data Flow Between Programs

```
[ 1 ]  MAINPGM initializes input & output variables (WS-NUM1, WS-NUM2, WS-TOTAL =
0)
        |
        ▼
[ 2 ]  MAINPGM issues CALL 'ADDPGM01' USING WS-NUM1, WS-NUM2, WS-TOTAL (BY
REFERENCE)
        |
        ▼
[ 3 ]  Control passes to ADDPGM01; LINKAGE SECTION maps NUM1/NUM2/TOTAL to the
same memory
        |
        ▼
[ 4 ]  ADDPGM01 computes TOTAL = NUM1 + NUM2, updating the shared memory directly
        |
        ▼
[ 5 ]  ADDPGM01 executes GOBACK → control (and the updated TOTAL) returns to
MAINPGM
        |
        ▼
[ 6 ]  MAINPGM DISPLAYs WS-TOTAL, which now reflects the result computed by the
sub-program
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between a Static CALL and a Dynamic CALL, both in syntax and compilation impact?
2. If PGMA calls PGMB statically, and PGMB calls PGMC dynamically, what needs to be recompiled if PGMC changes?
3. Why are static calls generally preferred for performance, and when would you deliberately choose a dynamic call instead?
4. What is the difference between CALL BY REFERENCE and CALL BY CONTENT?
5. How does a sub-program return a result to its calling program, given that COBOL has no explicit RETURN statement?
6. What goes in the LINKAGE SECTION of a sub-program, and why doesn't it allocate its own memory?
7. What typically causes an S0C4 or S0C7 abend related to a CALL statement?
8. Why should a sub-program never use STOP RUN?

9. Walk through the impact analysis you would perform before changing a widely-used sub-program in production.

Topic 10: String Handling Verbs

STRING, UNSTRING, INSPECT & Reference Modification

Definition

COBOL provides a dedicated set of verbs for manipulating alphanumeric (text) data: STRING to join multiple fields into one, UNSTRING to split one field into multiple, INSPECT to count or replace characters within a field, and reference modification to extract a substring directly by position and length.

Concept Explanation

STRING

STRING concatenates two or more source fields/literals into a single receiving field, optionally separating them with a DELIMITED BY clause, and tracks the ending position with a POINTER clause.

UNSTRING

UNSTRING is the reverse operation: it splits a single source field into multiple receiving fields, breaking on a specified delimiter (such as a comma), which is essential for parsing delimited input records.

INSPECT

INSPECT serves two purposes:

- TALLYING — counts how many times a character (or characters matching a condition, such as CHARACTERS, LEADING, or a specific literal) occurs in a field, storing the count in a counter variable.
- REPLACING — replaces occurrences of a character or string within a field with another value.

Finding the Actual (Trimmed) Length of a String

A field declared PIC X(20) always physically occupies 20 bytes, so LENGTH OF always returns 20 regardless of how much of it is meaningful data — it does not tell you where the real content ends and trailing spaces begin. To find the position of the last non-space character (the 'real' length), two common techniques are used:

33. INSPECT with FUNCTION REVERSE and TALLYING FOR LEADING SPACES — reverse the string, count the leading (originally trailing) spaces, then subtract that count from the field's defined length.
34. A loop using reference modification, scanning from the last byte backward until a non-space character is found.

(A naive approach — INSPECT ... TALLYING ... FOR CHARACTERS BEFORE INITIAL SPACE — only finds the position of the FIRST space, which is incorrect for any string that legitimately contains an internal space, such as 'ANIL KUMAR'; it would wrongly report a length of 4 instead of 10.)

Reference Modification (Substring)

Reference modification extracts a substring directly from a field using the syntax fieldname(starting-position:length), without needing a separate variable. For example, CUST-NAME(1:1) returns the first character, and CUST-NAME(1:2) returns the first two characters.

Example

```
      * ---------------- STRING -----------------------------
       01  WS-FIRST-NAME   PIC X(10) VALUE 'ANIL'.
       01  WS-LAST-NAME    PIC X(10) VALUE 'KUMAR'.
       01  WS-FULL-NAME    PIC X(21) VALUE SPACE.

           STRING WS-FIRST-NAME DELIMITED BY SPACE
                  ' '           DELIMITED BY SIZE
                  WS-LAST-NAME DELIMITED BY SPACE
                  INTO WS-FULL-NAME.
      *    Result: WS-FULL-NAME = 'ANIL KUMAR'

      * ---------------- UNSTRING ---------------------------
       01  WS-CSV-LINE     PIC X(30) VALUE 'ORD1111,ANIL,100000.00'.
       01  WS-ORDER-NUM    PIC X(10).
       01  WS-CUST-NAME    PIC X(10).
       01  WS-AMOUNT       PIC X(10).

           UNSTRING WS-CSV-LINE DELIMITED BY ','
               INTO WS-ORDER-NUM, WS-CUST-NAME, WS-AMOUNT.

      * ---------------- INSPECT TALLYING ----------------------
       01  WS-CUST-NAME2   PIC X(20) VALUE 'ANIL KUMAR'.
       01  WS-SPACE-CNT    PIC 9(02) VALUE ZERO.
       01  WS-STR-LEN      PIC 9(02) VALUE ZERO.

           INSPECT FUNCTION REVERSE(WS-CUST-NAME2)
               TALLYING WS-SPACE-CNT FOR LEADING SPACES.
           COMPUTE WS-STR-LEN = LENGTH OF WS-CUST-NAME2
                              - WS-SPACE-CNT.
      *    Result: WS-STR-LEN = 10

      * ---------------- INSPECT REPLACING ----------------------
           INSPECT WS-CUST-NAME2 REPLACING ALL 'ANIL' BY 'RAVI'.

      * ---------------- Reference Modification -----------------
           DISPLAY WS-CUST-NAME2(1:1).      *> First character
           DISPLAY WS-CUST-NAME2(1:4).      *> First 4 characters
```

Flowchart: Finding a Field's Actual (Trimmed) Length

```
[ 1 ]  Reverse the field using FUNCTION REVERSE
         |
```

```
        ▼
[ 2 ]  INSPECT the reversed field, TALLYING a counter FOR LEADING SPACES
        │
        ▼
[ 3 ]  COMPUTE Actual-Length = LENGTH OF Field – Trailing-Space-Count
        │
        ▼
[ 4 ]  Actual-Length now reflects the real content length, ignoring trailing
spaces
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between STRING and UNSTRING, and give a real scenario for each?
2. What are the two main uses of the INSPECT verb (TALLYING and REPLACING)?
3. How would you find the actual, meaningful length of a PIC X(20) field that contains trailing spaces?
4. Why is INSPECT ... TALLYING ... FOR CHARACTERS BEFORE INITIAL SPACE unreliable for names with an embedded space?
5. What is reference modification (substring) in COBOL, and what is its syntax?
6. What does the DELIMITED BY clause control in a STRING or UNSTRING statement?
7. What happens if the receiving fields in an UNSTRING are too small for the split data?
8. How would you check whether a specific character exists within a string field in COBOL?

Topic 11: Array / OCCURS Table

SEARCH & SEARCH ALL — PERFORM VARYING

Definition

The OCCURS clause is COBOL's mechanism for defining an array/table — a single data-name that represents multiple repetitions of the same data type and size in memory, accessed by a subscript or index. It is used whenever a program needs to hold multiple occurrences of the same kind of field or record.

Concept Explanation

Why Use OCCURS

Without OCCURS, storing 24 hourly temperature readings would require 24 separately named variables (TEMP-1, TEMP-2, ... TEMP-24), each needing its own hard-coded ACCEPT/DISPLAY/processing statement. OCCURS collapses that into a single variable name with 24 repetitions, so the same block of logic can process any occurrence just by changing a subscript — this is the foundation of using a loop instead of repeating code.

Types of Repetition

- Field repetition — the same elementary field repeated multiple times (e.g., 24 hourly temperature readings, or 6 subject marks for a student).
- Record repetition — an entire group/record repeated multiple times (e.g., 10 line-items on an invoice).

Static vs Dynamic (Variable) OCCURS

- Static OCCURS — OCCURS n TIMES; a fixed number of occurrences is always allocated. Commonly used for a fixed file layout (e.g., always exactly 24 hourly readings, or always exactly 13 monthly totals).
- Dynamic (Variable-length) OCCURS — OCCURS 1 TO n TIMES DEPENDING ON counter-field; the number of occurrences actually used varies based on a counter, though the maximum n is still fixed at compile time. Commonly used in WORKING-STORAGE for building a lookup/reference table from a file whose record count can change over time (e.g., a table of branch codes).

Accessing OCCURS Data — Subscript vs Index, and PERFORM VARYING

An occurrence is accessed by placing a position number in parentheses after the data-name, e.g., TEMP-READING(10). There are two mechanisms:

- Subscript — an ordinary WORKING-STORAGE numeric variable (e.g., WS-I) that the developer increments manually and uses as a position reference.
- Index — declared via INDEXED BY on the OCCURS clause; it is not a regular variable (it cannot be MOVEd into directly — it must be positioned using SET) and is required for the SEARCH/SEARCH ALL statements, where it acts internally as a displacement into the table for faster access.

PERFORM VARYING is the standard, compact way to loop through a table with a subscript or index, combining initialization, condition testing, and incrementing in a single statement: PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 24.

A subscript/index must always be initialized before first use, and the valid range must never be exceeded — accessing beyond the defined number of occurrences (e.g., occurrence 25 of a table defined for 24) does not raise a COBOL-level error by default and is a classic real-world cause of an S0C4 (invalid memory reference) abend.

Finding a Value in a Table — SEARCH vs SEARCH ALL

Both SEARCH and SEARCH ALL require the table to be defined with INDEXED BY, and both need the index reset (using SET index TO 1, not MOVE, since the index is not an ordinary variable) before each new search.

35. SEARCH — a sequential/linear search. It checks each occurrence one at a time, starting from the index's current position, and does NOT require the table data to be in any particular order. It is straightforward but, for a large table, can be slow because in the worst case it must check every occurrence.
36. SEARCH ALL — a binary search. It repeatedly checks the midpoint of the remaining range and eliminates half the remaining occurrences with each comparison, making it significantly faster than SEARCH for large tables. Its one strict requirement is that the table's data MUST already be sorted (ascending or descending, as declared) on the key being searched — SEARCH ALL will not work correctly, and may silently return wrong results, on unsorted data.

Where OCCURS Tables Are Used in Real Projects — Lookup Tables

A very common production pattern is a lookup/reference table: relatively small, largely static data (e.g., 50 branch codes and names) is either read once from a file into an OCCURS table at program start (used when the reference data changes fairly often), or hard-coded directly in the WORKING-STORAGE SECTION using FILLER/VALUE clauses that are then REDEFINED as an OCCURS table (used when the reference data changes rarely — any change requires editing and recompiling the copybook/program). A large, fast-changing dataset (e.g., a million-row daily transaction file) is never held entirely in an OCCURS table; instead, each transaction record is read one at a time and its key is looked up (via SEARCH/SEARCH ALL) against the smaller, in-memory reference table.

Example

```
      * ---- Field repetition, static OCCURS ----------------
       01  TEMP-DETAILS.
           03  TEMP-DATE       PIC X(10).
           03  TEMP-READING    OCCURS 24 TIMES
                               INDEXED BY TEMP-IDX
                                          PIC 9(02).

       01  WS-I                PIC 9(02) VALUE ZERO.

      * Loading readings using PERFORM VARYING with a subscript
```

```
           PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 24
               ACCEPT TEMP-READING(WS-I)
           END-PERFORM.

      * ---- Record repetition ---------------------------------
       01  INVOICE-BILL.
           03  INV-NUM          PIC X(08).
           03  ITEM-LIST OCCURS 10 TIMES INDEXED BY ITEM-IDX.
               05  ITEM-SNO     PIC 9(02).
               05  ITEM-NAME    PIC X(15).
               05  ITEM-QTY     PIC 9(03).
               05  ITEM-PRICE   PIC 9(07)V99.

      * ---- SEARCH (linear) -----------------------------------
           SET TEMP-IDX TO 1.
           SEARCH TEMP-READING
               AT END
                   DISPLAY 'TEMPERATURE OF 20 NOT FOUND'
               WHEN TEMP-READING(TEMP-IDX) = 20
                   DISPLAY '20 DEGREES FOUND AT HOUR ' TEMP-IDX
           END-SEARCH.

      * ---- SEARCH ALL (binary — table must be sorted) ---------
           SET TEMP-IDX TO 1.
           SEARCH ALL TEMP-READING
               AT END
                   DISPLAY 'VALUE NOT FOUND'
               WHEN TEMP-READING(TEMP-IDX) = 20
                   DISPLAY 'FOUND AT HOUR ' TEMP-IDX
           END-SEARCH.
```

Flowchart: SEARCH (Linear) vs SEARCH ALL (Binary)

```
[ 1 ]  SEARCH: start at index 1 → compare occurrence 1, then 2, then 3 in order
        |
        ▼
[ 2 ]          until a match is found, or AT END is reached after checking every
occurrence
        |
        ▼
[ 3 ]  SEARCH ALL: table MUST be pre-sorted on the search key
        |
        ▼
[ 4 ]  Check the MIDDLE occurrence → match? Done.
        |
        ▼
[ 5 ]  Value is lower → discard the upper half, repeat on the lower half
        |
```

```
        ▼
[ 6 ]  Value is higher → discard the lower half, repeat on the upper half
        │
        ▼
[ 7 ]  Continue halving the remaining range until found, or the range is empty (AT
END)
```

Flowchart: Building & Using a Lookup Table Against a Large Transaction File

```
[ 1 ]  Load the small reference file (e.g., 50 branch codes) once into an OCCURS
table at program start
        │
        ▼
[ 2 ]  Open and read the large transaction file, one record at a time
        │
        ▼
[ 3 ]  For each transaction record, SEARCH/SEARCH ALL the in-memory reference
table for a matching key
        │
        ▼
[ 4 ]  Found → enrich/validate the transaction using the reference data
        │
        ▼
[ 5 ]  Not found → route the transaction to an error/exception path
        │
        ▼
[ 6 ]  Read the next transaction record; repeat until end-of-file
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between field repetition and record repetition when using OCCURS?
2. What is the difference between a static OCCURS and a variable/dynamic OCCURS (OCCURS ... DEPENDING ON)?
3. What is the difference between a subscript and an index, and why does SEARCH/SEARCH ALL require an index?
4. Explain the difference between SEARCH and SEARCH ALL, including their performance characteristics and prerequisites.
5. Why must the table be sorted before using SEARCH ALL, and what happens if it isn't?
6. How would you loop through an OCCURS table using PERFORM VARYING?
7. What is a common real-world cause of an S0C4 abend when working with OCCURS tables?

8. Describe a lookup-table design you would use to validate a large transaction file against a small reference/master file.
9. If a static OCCURS table is defined for 20 occurrences but the input file unexpectedly contains 21 records, what happens, and how would you troubleshoot it in production?
10. What is the difference between storing reference data in a hard-coded REDEFINES/OCCURS block in WORKING-STORAGE versus loading it from a file at runtime?

Appendix: Quick-Reference — Common S0C4 / S0C7 Abend Causes

Definition

An S0C4 abend is a Program Interrupt caused by an invalid memory reference (accessing storage the program is not entitled to access). An S0C7 abend is caused by invalid or non-numeric data being used in a numeric operation (e.g., COMPUTE/ADD on a field that does not contain valid packed/zoned decimal data). Both are extremely common in mainframe production support and are frequently discussed in interviews for support-focused roles.

Concept Explanation — Frequent Root Causes Seen in Production

Data-Related Causes (most common in day-to-day support)

- Indexing or subscripting outside the defined limits of an OCCURS table (e.g., a table defined for 20 occurrences receiving a 21st input record).
- Bad/non-numeric data landing in a field defined as numeric (typically an upstream data-quality issue, causing S0C7).

Developer/Coding-Related Causes (root-caused back to the program logic itself)

- Referencing a file's record area before the file has been OPENed.
- Missing or incorrect DD statement in the JCL for a file the program expects.
- LINKAGE SECTION coded incorrectly, or referenced without the corresponding USING clause on the PROCEDURE DIVISION or ENTRY statement.
- Mismatched parameter list (count, order, or size) between the calling program's CALL...USING and the sub-program's PROCEDURE DIVISION USING.
- STOP RUN or GO TO coded improperly inside a SORT input/output procedure, or a PERFORM/GO TO that jumps outside the boundary of a SORT input/output procedure.

In an interview setting, it is generally best to lead with the data-related and impact-analysis angle (how you diagnosed the issue and what the business impact/fix was) rather than dwelling on the specific coding mistake, since the objective in support work is resolving the incident and preventing recurrence.

Flowchart: Typical Production Support Triage for an Abend

```
[ 1 ]  Job abends -> note the Abend Code (e.g., S0C4/S0C7), Step name, and Offset
in the load module
        |
        ▼
[ 2 ]  Open the SYSOUT / job log and the system dump to identify the failing
paragraph and statement
        |
        ▼
[ 3 ]  Check the record/field being processed at the point of failure for bad or
unexpected data
```

```
        |
        ▼
[ 4 ]  Classify: Data issue (bad input) vs Program issue (logic/coding defect) vs
Environment issue (JCL/DD/file)
        |
        ▼
[ 5 ]  Apply the appropriate fix: correct/exclude the bad data and rerun, or raise
a code-change ticket, or fix the JCL
        |
        ▼
[ 6 ]  Document root cause and resolution; escalate per priority (e.g., P1 = 2-4
hrs, P2 = 4-8 hrs) if within SLA
```

Possible Interview Questions

Possible Interview Questions (4+ Year Mainframe Developer / Support Engineer)

1. What is the difference between an S0C4 and an S0C7 abend?
2. Walk me through how you would triage and resolve a production S0C4 abend.
3. What information would you look for first when a job abends - abend code, offset, dump, or job log - and why?
4. Give an example of a data-related cause versus a coding-related cause of an S0C4 abend.
5. How would you distinguish between an issue caused by bad input data versus a genuine program defect?
6. What is a P1 versus a P2 production incident, and how does that affect your response time?
7. How would you prevent a recurring abend from happening again after a one-time fix?